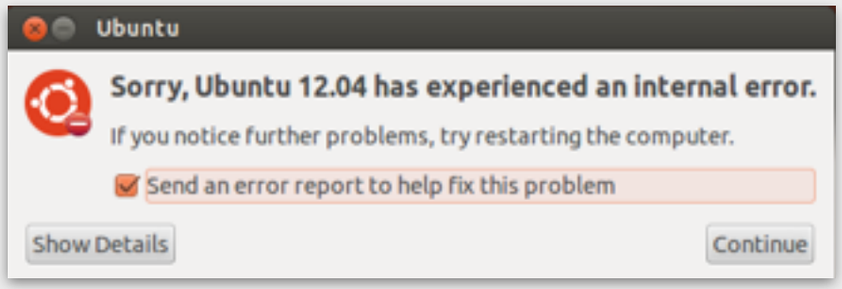
Evolutionary Testing for **Crash** Reproduction

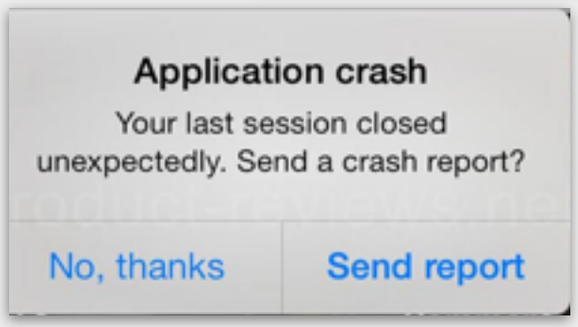
Mozhan Soltani

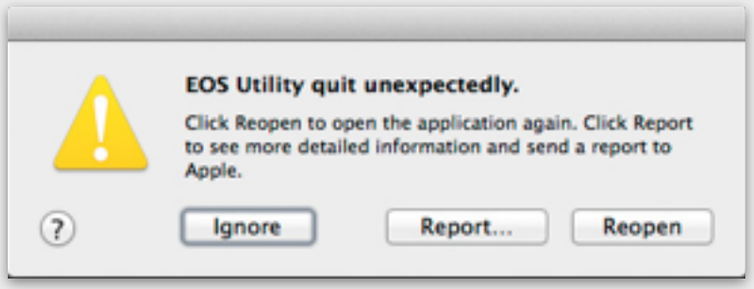
Annibale Panichella

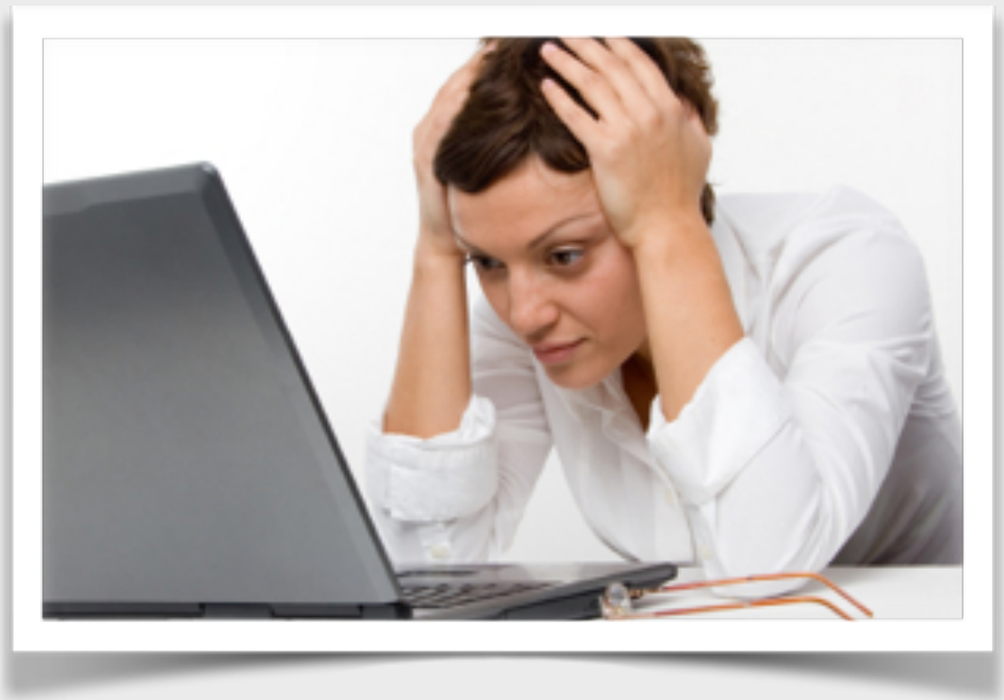
Arie van Deursen

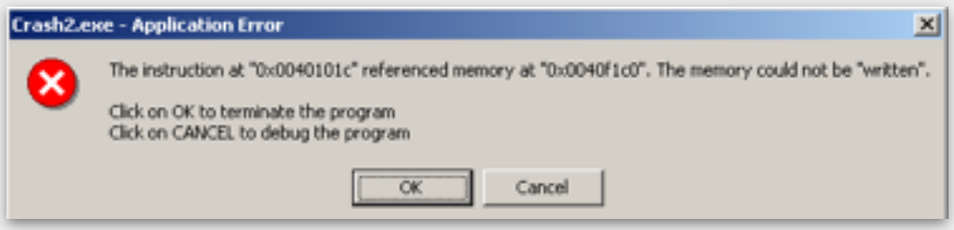
Bugs are everywhere…

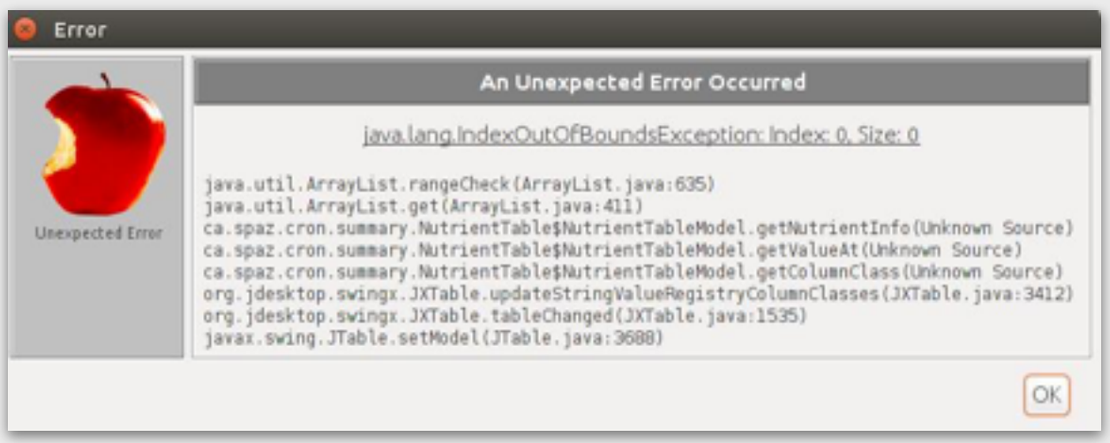




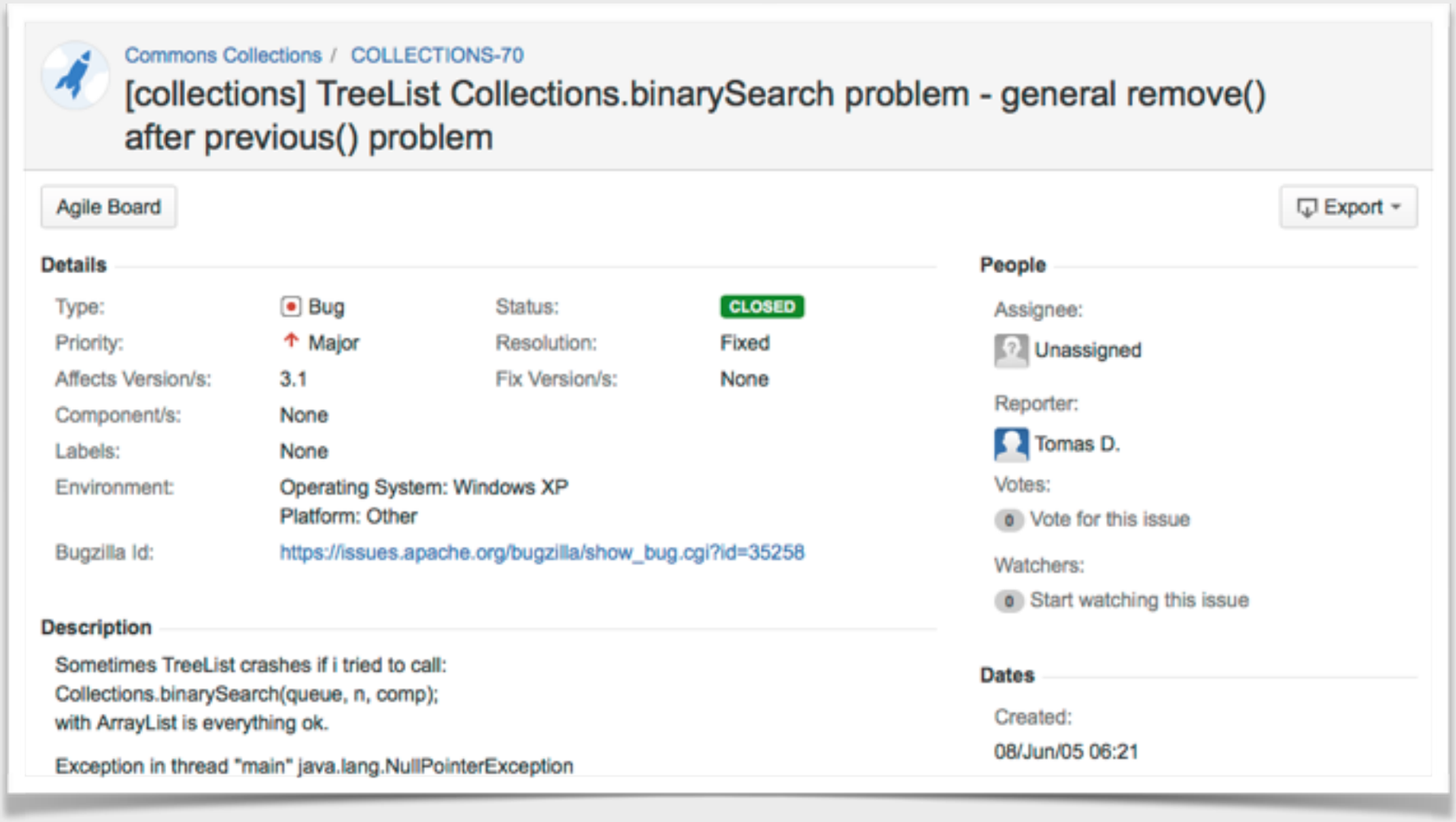








Cost of Bug Fixings



Commons Collections / COLLECTIONS-70

[collections] TreeList Collections.binarySearch problem - general remove() after previous() problem

Agile Board

Export

Details

Type:	Bug	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	3.1	Fix Version/s:	None
Component/s:	None		
Labels:	None		
Environment:	Operating System: Windows XP Platform: Other		
Bugzilla Id:	https://issues.apache.org/bugzilla/show_bug.cgi?id=35258		

Description

Sometimes TreeList crashes if i tried to call:
Collections.binarySearch(queue, n, comp);
with ArrayList is everything ok.

Exception in thread "main" java.lang.NullPointerException

People

Assignee: Unassigned

Reporter: Tomas D.

Votes: 0 Vote for this issue

Watchers: 0 Start watching this issue

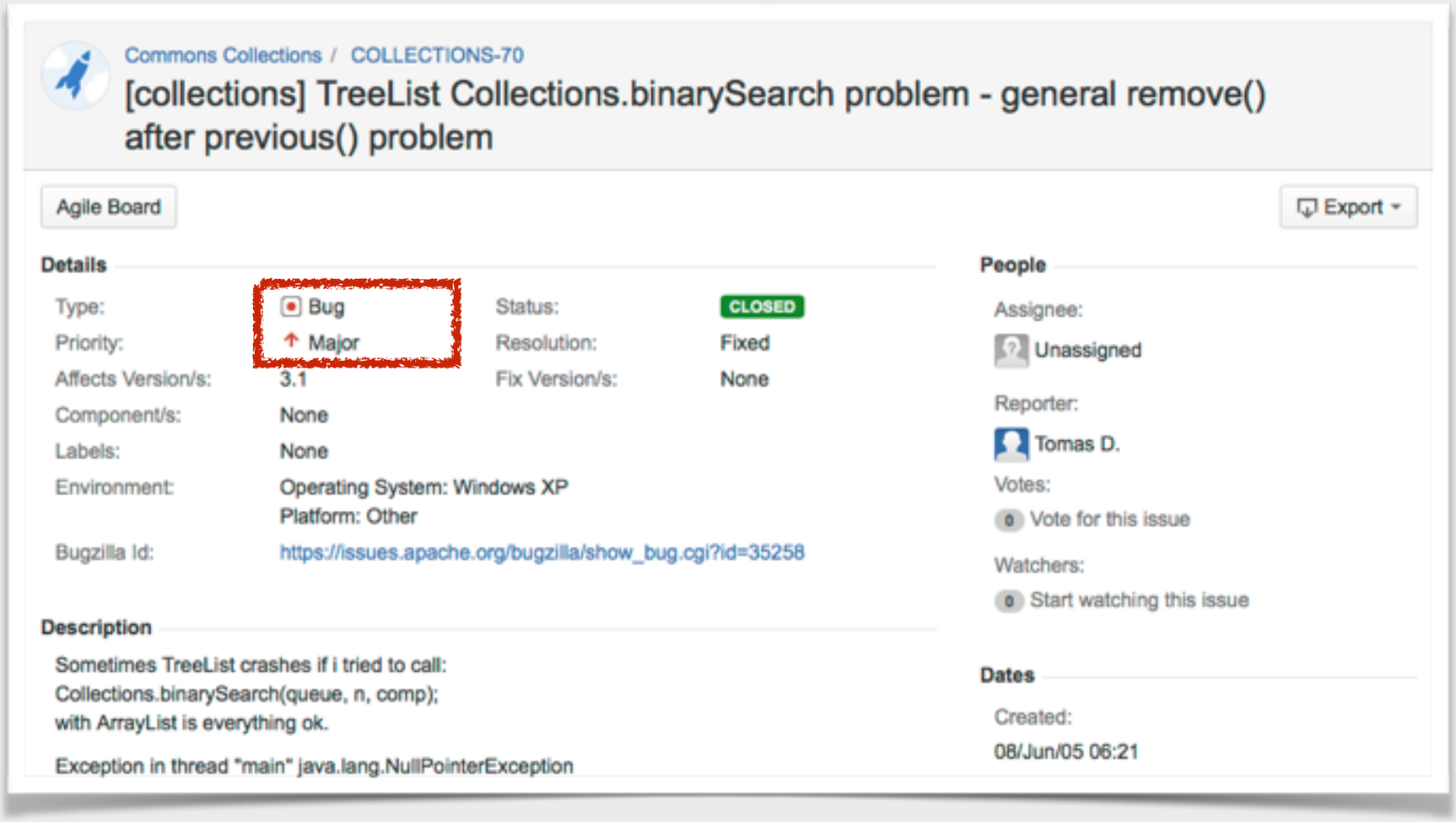
Dates

Created: 08/Jun/05 06:21

Major Bug for Apache Commons Collections

https://issues.apache.org/jira/browse/COLLECTIONS-70

Cost of Bug Fixings



Commons Collections / COLLECTIONS-70

[collections] TreeList Collections.binarySearch problem - general remove() after previous() problem

Agile Board

Export

Details

Type:	Bug	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	3.1	Fix Version/s:	None
Component/s:	None		
Labels:	None		
Environment:	Operating System: Windows XP Platform: Other		
Bugzilla Id:	https://issues.apache.org/bugzilla/show_bug.cgi?id=35258		

Description

Sometimes TreeList crashes if i tried to call:
Collections.binarySearch(queue, n, comp);
with ArrayList is everything ok.

Exception in thread "main" java.lang.NullPointerException

People

Assignee: Unassigned

Reporter: Tomas D.

Votes: 0 Vote for this issue

Watchers: 0 Start watching this issue

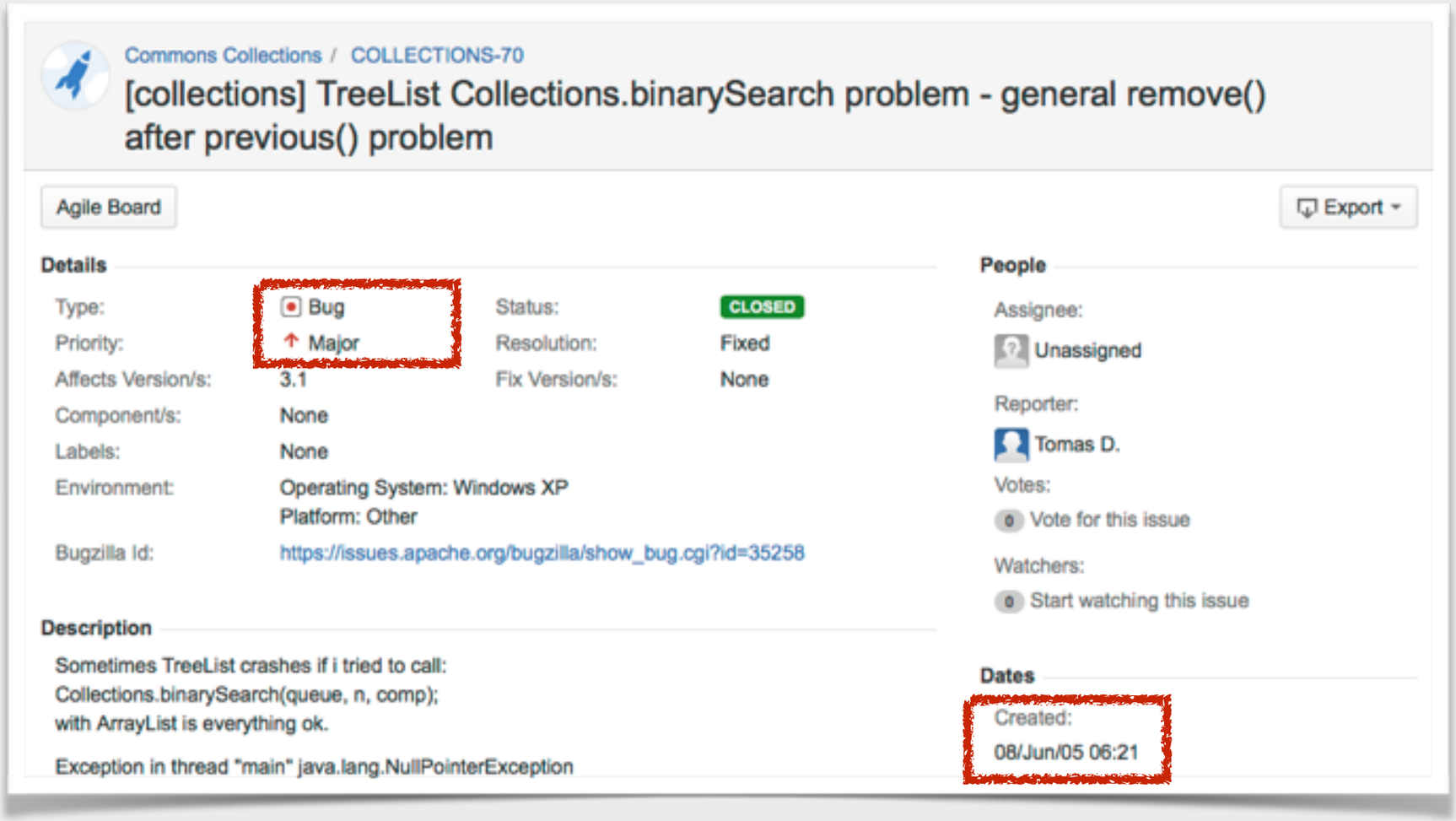
Dates

Created: 08/Jun/05 06:21

Major Bug for Apache Commons Collections

https://issues.apache.org/jira/browse/COLLECTIONS-70

Cost of Bug Fixings



Commons Collections / COLLECTIONS-70

[collections] TreeList Collections.binarySearch problem - general remove() after previous() problem

Agile Board

Export

Details

Type:	Bug	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	3.1	Fix Version/s:	None
Component/s:	None		
Labels:	None		
Environment:	Operating System: Windows XP Platform: Other		
Bugzilla Id:	https://issues.apache.org/bugzilla/show_bug.cgi?id=35258		

Description

Sometimes TreeList crashes if i tried to call:
Collections.binarySearch(queue, n, comp);
with ArrayList is everything ok.

Exception in thread "main" java.lang.NullPointerException

People

Assignee: Unassigned

Reporter: Tomas D.

Votes: 0 Vote for this issue

Watchers: 0 Start watching this issue

Dates

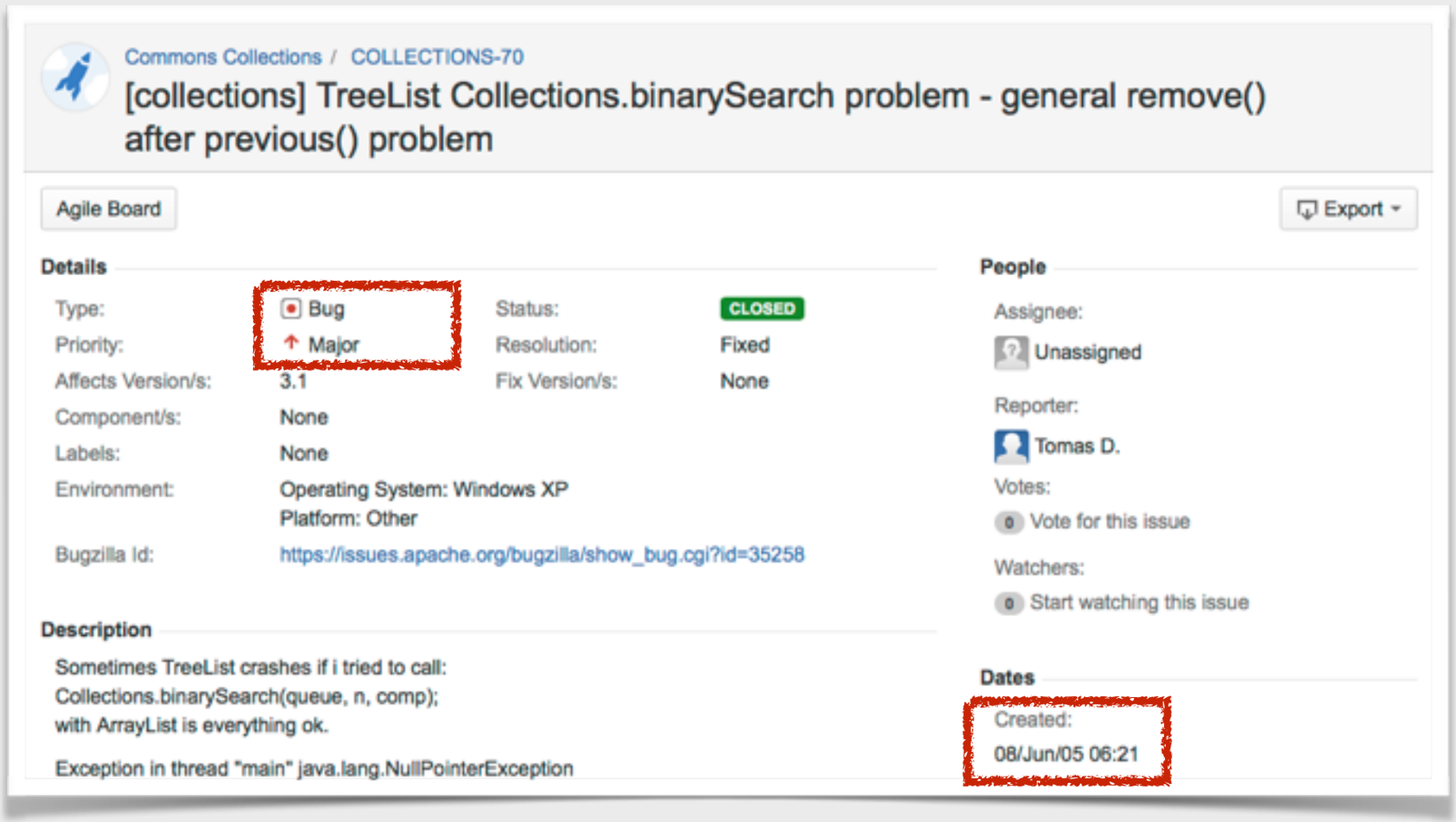
Created: 08/Jun/05 06:21

Major Bug for Apache Commons Collections

Created on June 2005

https://issues.apache.org/jira/browse/COLLECTIONS-70

Cost of Bug Fixings


Commons Collections / COLLECTIONS-70

[collections] TreeList Collections.binarySearch problem - general remove() after previous() problem

Agile Board

Export

Details

Type: Bug
Priority: Major
Affects Version/s: 3.1
Component/s: None
Labels: None
Environment: Operating System: Windows XP
Platform: Other
Bugzilla Id: https://issues.apache.org/bugzilla/show_bug.cgi?id=35258

Status: CLOSED
Resolution: Fixed
Fix Version/s: None

People

Assignee: Unassigned
Reporter: Tomas D.
Votes: 0 Vote for this issue
Watchers: 0 Start watching this issue

Description

Sometimes TreeList crashes if i tried to call:
Collections.binarySearch(queue, n, comp);
with ArrayList is everything ok.

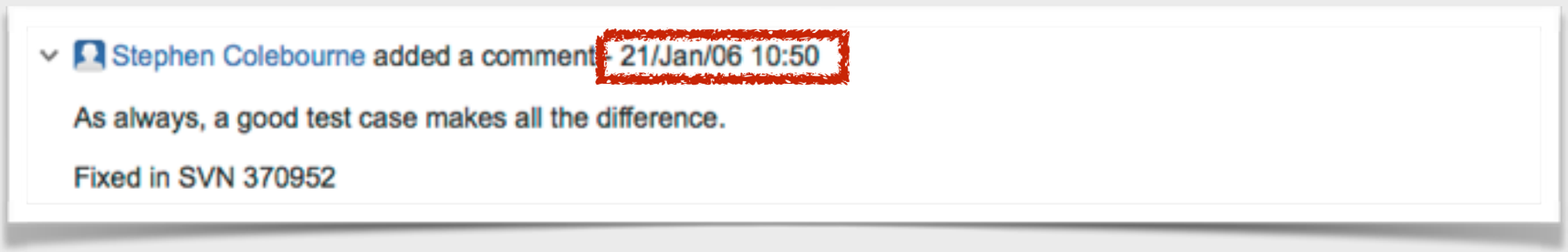
Exception in thread "main" java.lang.NullPointerException

Dates

Created: 08/Jun/05 06:21

Major Bug for Apache Commons Collections

Created on June 2005


Stephen Colebourne added a comment - 21/Jan/06 10:50

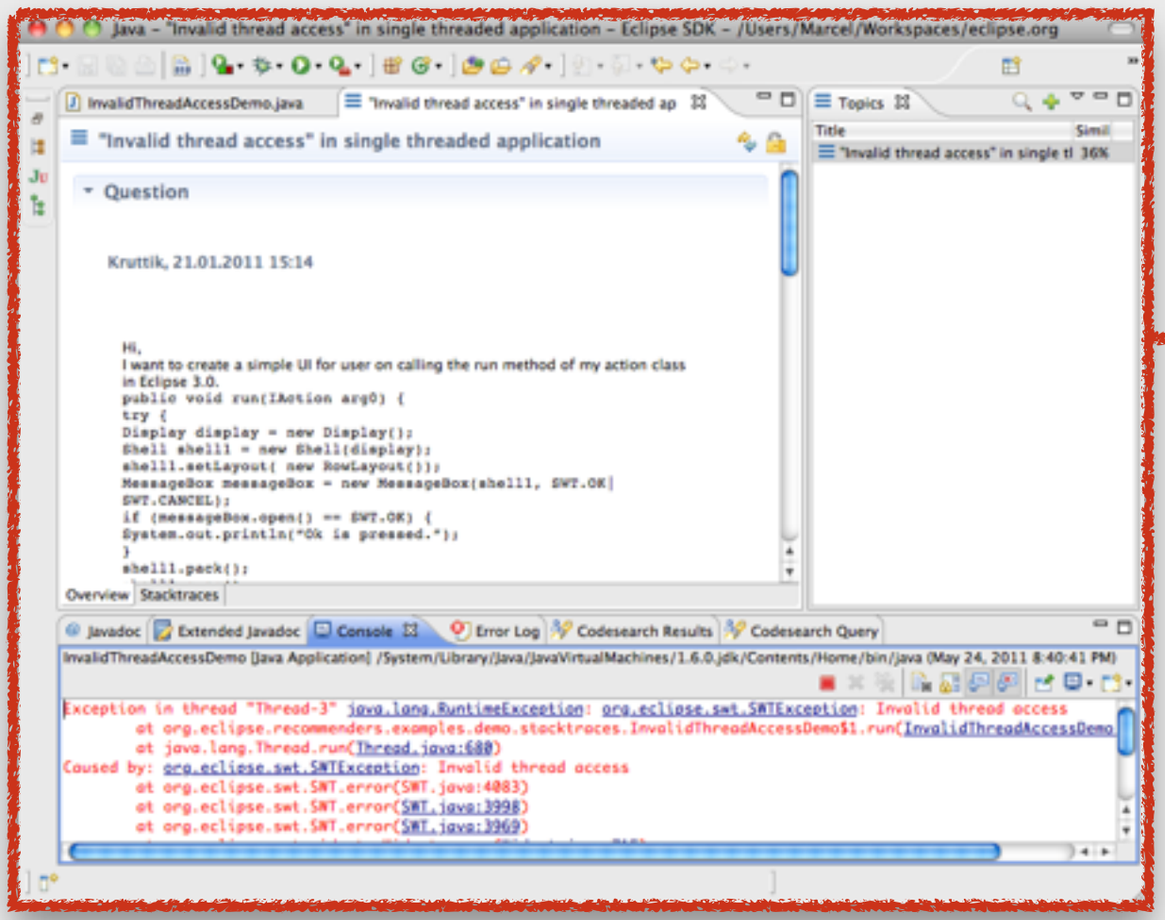
As always, a good test case makes all the difference.

Fixed in SVN 370952

Solved on January 2006

https://issues.apache.org/jira/browse/COLLECTIONS-70

Crash Replication



1) Inspect the stack trace / bug report
2) Analyse the code
3) **Replicate the crash**
4) Fix the problem

Related Work

Record-and-play approaches

ReCrashJ: a Tool for Capturing and Reproducing Program Crashes in Deployed Applications

Shay Artzi, IBM T.J. Watson Research Center, artzi@us.ibm.com
Sunghun Kim, University of Hong Kong Science and Technology, hunkim@cse.ust.hk
Michael D. Ernst, University of Washington, mernst@cs.washington.edu

ABSTRACT

Many programs have latent bugs that cause the program to fail. In order to fix a failing program, it is crucial to be able to reproduce the failure consistently. However, reproducing a failure can be difficult and time consuming, especially when the failure is discovered by a user in a deployed application.

We present ReCrash, an approach to reproduce failures efficiently, both locally and in deployed applications, without any changes to the host's environment, and with low execution overhead.

During execution, ReCrash efficiently stores part of the state of method arguments. If the program fails, ReCrash uses the stored information to create unit tests that reproduce the failure. This is effective because programs written in object-oriented style rely mostly on near-by state.

This demo presents ReCrashJ, an implementation of ReCrash for Java. We show the ReCrashJ Eclipse plug-in (for developers) and the ReCrashJ command-line modules (for deployed software).

Categories and Subject Descriptors

D2.5 [Testing and Debugging]: Monitors and Tracing; F3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Theory

Keywords

crash, monitoring, tracing, unit test, reproducing

1. Introduction

It is difficult to find and eliminate a software failure, and especially to verify a solution, without the ability to consistently reproduce the failure. This demo presents ReCrash, a technique that simplifies the task of reproducing failures. ReCrash reproduces failures that were discovered in deployed applications.

ReCrash automatically converts a failing program execution into a set of deterministic, self-contained unit tests. Each of the unit tests reproduces the same failure by starting the execution from a different snapshot (checkpoint) of the system taken during the execution.

Reproducing a failure found in a deployed application is hard, even when program inputs, environmental variables, and GUI actions are available. Existing techniques either require changes to the host environment [10], [7], [11] or supply only one snapshot (checkpoint) of the system before the reproducible failure [3, 6, 9, 12]. Our technique requires no changes to the host environment and supplies the developer with multiple snapshots of the system before the failure. Execution from each snapshot leads to the original observed failure. Having multiple snapshots of the system can be useful in finding the underlying bug, as the developer might get a better view of the cause of failure. In addition, some snapshots allow the developer to debug the failure quickly using a shorter execution instead of a longer execution that exposes the same failure.

2. ReCrash

This section briefly describes the ReCrash technique, implementation, optimizations, limitations, and experimental results. More details appear in our ECOOP 2008 paper [1].

2.1 Technique

ReCrash is based on the following characteristics of object-oriented programs: bugs are dependent on small parts of the heap, important state is encapsulated nearby, and global information is not used excessively. Thus, in order to reproduce a method's execution it might not be necessary to store the entire transitive state of the methods' arguments on entry.

ReCrash has two phases: monitoring and test case generation. In the monitoring phase, ReCrash maintains a shadow call-stack containing partial state of the arguments to the methods on the original call-stack. The part of the transitive state of the arguments that is not copied into the shadow stack refers to the original objects on the heap. When the program fails (i.e., crashes), ReCrash serializes the shadow stack contents, including all heap objects referred to from the shadow stack.

In the test generation phase, ReCrash generates candidate tests by calling methods from the de-serialized shadow call stack. Each test executes the original method using the de-serialized receiver and arguments (stored at the time of the crash). ReCrash filters all test candidate that do not reproduce the original failure.

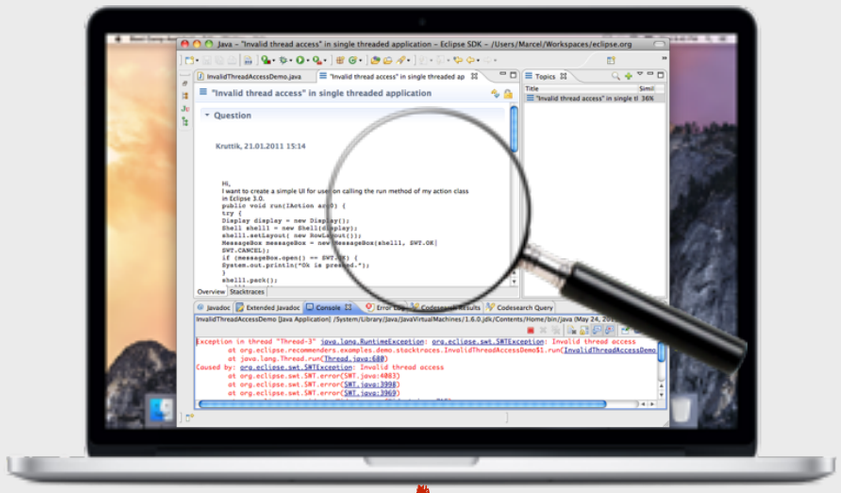
ReCrash outputs multiple tests to create a better view of the failure for the developer. For example, tests that call methods from the top of the call stack may not provide enough context to find the error, while tests that call methods from the bottom of the call-stack provide more context, but are less likely to reproduce the original failure and may contain extraneous, distracting details.

2.2 Optimizations

ReCrash's time and space overheads are mostly determined by

ESEC-FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.
ACM 978-1-60558-001-2/09/08.

S. Artzi et al., ESEC/FSE 2009.



Data from Heap and Stack

Stored Objects

```
@Test
public void test(){
    boolean boolean0 = false;
    String string0 = BooleanUtils.toStringOnOff(boolean0);
    assertNotNull(string0);
    assertEquals("off", string0);
}
```

Related Work

Record-and-play approaches

ReCrashJ: a Tool for Capturing and Reproducing Program Crashes in Deployed Applications

Shay Artzi
IBM T.J. Watson Research Center
artzi@us.ibm.com

Sunghun Kim
University of Hong Kong Science and Technology
hunkim@cse.ust.hk

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

ABSTRACT

Many programs have latent bugs that cause the program to fail. In order to fix a failing program, is it crucial to be able to reproduce the failure consistently. However, reproducing a failure can be difficult and time consuming, especially when the failure is discovered by a user in a deployed application.

We present ReCrash, an approach to reproduce failures efficiently, both locally and in deployed applications, without any changes to the host's environment, and with low execution overhead.

During execution, ReCrash efficiently stores part of the state of method arguments. If the program fails, ReCrash uses the stored information to create unit tests that reproduce the failure. This is effective because programs written in object-oriented style rely mostly on near-by state.

This demo presents ReCrashJ, an implementation of ReCrash for Java. We show the ReCrashJ Eclipse plug-in (for developers) and the ReCrashJ command-line modules (for deployed software).

Categories and Subject Descriptors

D2.5 [**Testing and Debugging**]: Monitors and Tracing; F3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Theory

Keywords

crash, monitoring, tracing, unit test, reproducing

1. Introduction

It is difficult to find and eliminate a software failure, and especially to verify a solution, without the ability to consistently reproduce the failure. This demo presents ReCrash, a technique that simplifies the task of reproducing failures. ReCrash reproduces failures that were discovered in deployed applications.

ReCrash automatically converts a failing program execution into a set of deterministic, self-contained unit tests. Each of the unit tests reproduces the same failure by starting the execution from a different snapshot (checkpoint) of the system taken during the execution.

Reproducing a failure found in a deployed application is hard, even when program inputs, environmental variables, and GUI actions are available. Existing techniques either require changes to the host environment [10], [7], [11] or supply only one snapshot (checkpoint) of the system before the reproducible failure [3, 6, 9, 12]. Our technique requires no changes to the host environment and supplies the developer with multiple snapshots of the system before the failure. Execution from each snapshot leads to the original observed failure. Having multiple snapshots of the system can be useful in finding the underlying bug, as the developer might get a better view of the cause of failure. In addition, some snapshots allow the developer to debug the failure quickly using a shorter execution instead of a longer execution that exposes the same failure.

2. ReCrash

This section briefly describes the ReCrash technique, implementation, optimizations, limitations, and experimental results. More details appear in our ECOOP 2008 paper [1].

2.1 Technique

ReCrash is based on the following characteristics of object oriented programs: bugs are dependent on small parts of the heap, important state is encapsulated nearby, and global information is not used excessively. Thus, in order to reproduce a method's execution it might not be necessary to store the entire transitive state of the methods' arguments on entry.

ReCrash has two phases: monitoring and test case generation. In the monitoring phase, ReCrash maintains a shadow call-stack containing partial state of the arguments to the methods on the original call-stack. The part of the transitive state of the arguments that is not copied into the shadow stack refers to the original objects on the heap. When the program fails (i.e., crashes), ReCrash serializes the shadow stack contents, including all heap objects referred to from the shadow stack.

In the test generation phase, ReCrash generates candidate tests by calling methods from the de-serialized shadow call stack. Each test executes the original method using the de-serialized receiver and arguments (stored at the time of the crash). ReCrash filters all test candidate that do not reproduce the original failure.

ReCrash outputs multiple tests to create a better view of the failure for the developer. For example, tests that call methods from the top of the call stack may not provide enough context to find the error, while tests that call methods from the bottom of the call-stack provide more context, but are less likely to reproduce the original failure and may contain extraneous, distracting details.

2.2 Optimizations

ReCrash's time and space overheads are mostly determined by

ESEC-FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.
ACM 978-1-60558-001-2/09/08.

S. Artzi et al., ESEC/FSE 2009.

Disadvantages:

1) Require software instrumentation and special hardware deployment

2) Memory and run-time overhead:
   - 10%-90% of memory overhead
   - 30%-60% of runtime overhead

Related Work

Core dump-based approaches

Reconstructing Core Dumps

Jeremias Rößler* · Andreas Zeller* · Gordon Fraser† · Cristian Zamfir‡ · George Candea‡
*Saarland University, Saarbrücken, Germany
Email: {roessler, zeller}@cs.uni-saarland.de
†University of Sheffield, Sheffield, United Kingdom
Email: gordon.fraser@sheffield.ac.uk
‡École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
Email: {cristian.zamfir, george.candea}@epfl.ch

Abstract—When a software failure occurs in the field, it is often difficult to reproduce. Guided by a memory dump at the moment of failure (a "core dump"), our RECORE test case generator searches for a series of events that precisely reconstruct the failure from primitive data. Applied on seven non-trivial Java bugs, RECORE reconstructs the exact failure in five cases without any runtime overhead in production code.

Keywords-debugging; failure reproduction; test case generation; memory dumps

I. Introduction

When a program fails in the field, the developer who wants to debug the failure must first be able to *reproduce* it. Being able to reproduce a failure locally is important for diagnostic purposes, because developers can then inspect arbitrary aspects of the failing execution to locate the failure cause. If the problem cannot be reproduced, though, one can never know whether the fix is successful: Only if the failure goes away after the fix has been applied does the developer know that she effectively found the failure cause.

Reproducing a failure is easy—in theory. Since a program is but a mathematical mapping of inputs to outputs, all it takes is to record the inputs to reproduce the output. The problem in practice is that recording inputs incurs time and memory overhead, and neither developers nor users may be willing to spend resources on a diagnostic feature that (in the best of cases) will never be used.

Once a failure *has* occurred, though, such diagnostic overhead no longer matters. A common practice on modern operating systems is to produce a *core dump*—a persistent snapshot of the stack and heap memory at the moment of the failure. Such a core dump can be loaded into interactive debuggers, and programmers can then inspect it post-mortem to understand how the failure came to be.

As an example, consider Figure 1, showing a stack trace originating from a failure in the Java JodaTime date and time library. We see that the `toInterval()` method was called with some specific date and time zone, resulting in the method `localToUTC()` raising an `IllegalArgumentException`. In addition to this stack trace, a full core dump would also hold all values on the stack and on the heap at the moment of failure.

Such a core dump, however, just reports the final state, not the history. Assuming one finds an illegal value in some variable: How did that value come to be? The input that caused the failure may long have been overwritten by later input. What was it that caused the problem in the first place? With a core dump, you are a detective faced with a *corpse*—and you have to reconstruct all the events that lead to it.

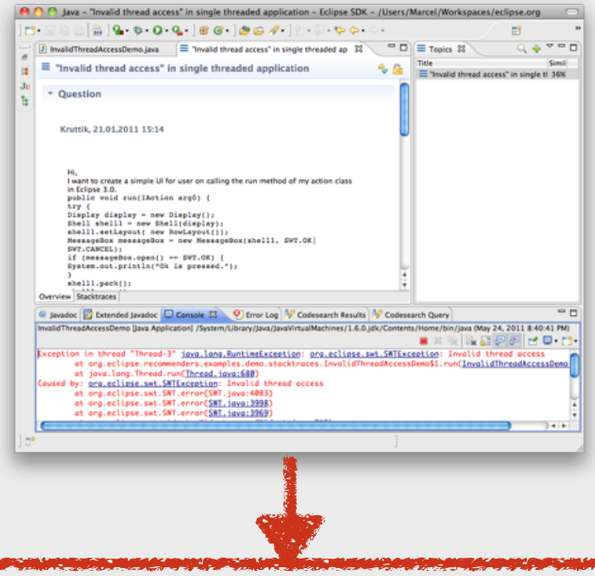
For computer programs, however, we do have tools that can construct such events automatically. *Test case generation* addresses the problem of generating inputs that fulfill a number of constraints—typically, reaching a particular location in the program. *Constraint-based test generators* do so by solving the path conditions from input to these locations. Reproducing a full core dump, though, would impose far too many constraints on any practical constraint solver.

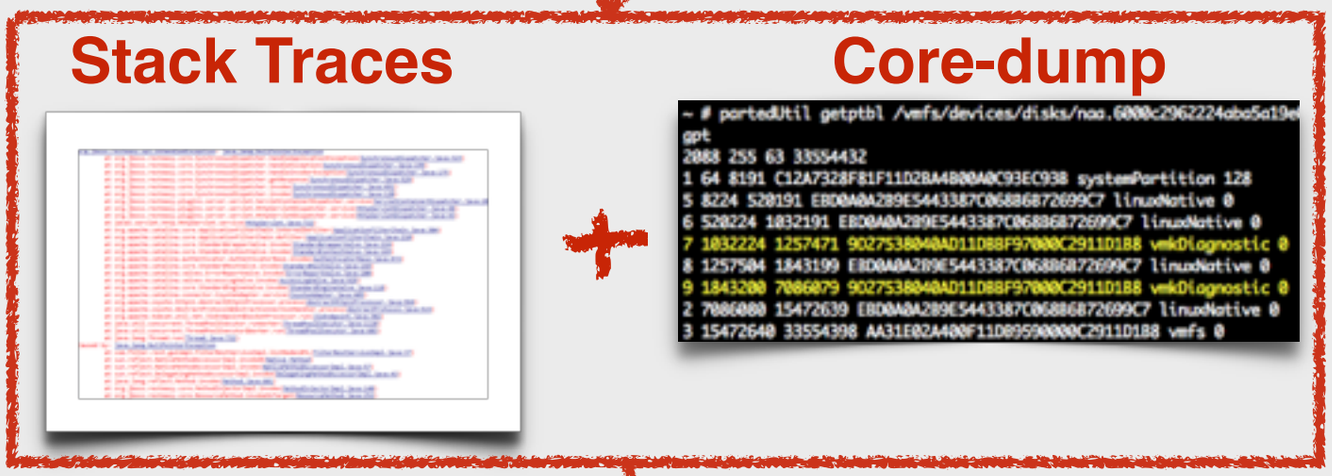
In practice, it may suffice to have a test case that produces a state *as similar as possible* to the core dump. This is the domain of *search-based test generators*, which systematically evolve the input according to a fitness function. If similarity to the core dump induces high fitness, we could then *automatically obtain a test case that reproduces the core dump*—without incurring overhead during execution.

This is what we explore in this paper. Given a stack and heap dump, our RECORE prototype uses search-based test generation to automatically produce a set of inputs that

Figure 1. JodaTime bug 2487417: `toInterval()` fails when processing October 18, 2009, Brasil time.

J. Rößler et al., ICST 2013.






```
@Test
public void test(){
    boolean boolean0 = false;
    String string0 = BooleanUtils.toStringOnOff(boolean0);
    assertNotNull(string0);
    assertEquals("off", string0);
}
```

Related Work

Core dump-based approaches

Reconstructing Core Dumps

Jeremias Rößler* · Andreas Zeller* · Gordon Fraser† · Cristian Zamfir‡ · George Candea‡
*Saarland University, Saarbrücken, Germany
Email: {roessler, zeller}@cs.uni-saarland.de
†University of Sheffield, Sheffield, United Kingdom
Email: gordon.fraser@sheffield.ac.uk
‡École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
Email: {cristian.zamfir, george.candea}@epfl.ch

Abstract—When a software failure occurs in the field, it is often difficult to reproduce. Guided by a memory dump at the moment of failure (a "core dump"), our RECORE test case generator searches for a series of events that precisely reconstruct the failure from primitive data. Applied on seven non-trivial Java bugs, RECORE reconstructs the exact failure in five cases without any runtime overhead in production code.

Keywords-debugging; failure reproduction; test case generation; memory dumps

I. Introduction

When a program fails in the field, the developer who wants to debug the failure must first be able to *reproduce* it. Being able to reproduce a failure locally is important for diagnostic purposes, because developers can then inspect arbitrary aspects of the failing execution to locate the failure cause. If the problem cannot be reproduced, though, one can never know whether the fix is successful: Only if the failure goes away after the fix has been applied does the developer know that she effectively found the failure cause.

Reproducing a failure is easy—in theory. Since a program is but a mathematical mapping of inputs to outputs, all it takes is to record the inputs to reproduce the *output*. The problem in practice is that recording inputs incurs time and memory overhead, and neither developers nor users may be willing to spend resources on a diagnostic feature that (in the best of cases) will never be used.

Once a failure *has* occurred, though, such diagnostic overhead no longer matters. A common practice on modern operating systems is to produce a *core dump*—a persistent snapshot of the stack and heap memory at the moment of the failure. Such a core dump can be loaded into interactive debuggers, and programmers can then inspect it post-mortem to understand how the failure came to be.

As an example, consider Figure 1, showing a stack trace originating from a failure in the Java JodaTime date and time library. We see that the `toInterval()` method was called with some specific date and time zone, resulting in the method `localToUTC()` raising an `IllegalArgumentException`. In addition to this stack trace, a full core dump would also hold all values on the stack and on the heap at the moment of failure.

Such a core dump, however, just reports the final state, not the history. Assuming one finds an illegal value in some variable: How did that value come to be? The input that caused the failure may long have been overwritten by later input. What was it that caused the problem in the first place? With a core dump, you are a detective faced with a corpse—and you have to reconstruct all the events that lead to it.

For computer programs, however, we do have tools that can construct such events automatically. *Test case generation* addresses the problem of generating inputs that fulfill a number of constraints—typically, reaching a particular location in the program. *Constraint-based test generators* do so by solving the path conditions from input to these locations. Reproducing a full core dump, though, would impose far too many constraints on any practical constraint solver.

In practice, it may suffice to have a test case that produces a state *as similar as possible* to the core dump. This is the domain of *search-based test generators*, which systematically evolve the input according to a fitness function. If similarity to the core dump induces high fitness, we could then *automatically obtain a test case that reproduces the core dump*—without incurring overhead during execution.

This is what we explore in this paper. Given a stack and heap dump, our RECORE prototype uses search-based test generation to automatically produce a set of inputs that

Figure 1. JodaTime bug 2387437: `toInterval()` fails when processing October 18, 2009, Brasil time.

J. Rößler et al., ICST 2013.

Advantages:

1) No overhead due to system monitoring
2) SBST Approaches

Disadvantages:

1) Requires core-dump at the time of the crash in addition to stack traces
2) Ability to replicate crashes depends on the amount of core-dump data available

Stack Trace based Crash Replication

Symbolic Execution

STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution

Ning Chen and Sunghun Kim, *Member, IEEE*

Abstract—Software crash reproduction is the necessary first step for debugging. Unfortunately, crash reproduction is often labor intensive. To automate crash reproduction, many techniques have been proposed including record-replay and post-failure-process approaches. Record-replay approaches can reliably replay recorded crashes, but they incur substantial performance overhead to program executions. Alternatively, post-failure-process approaches analyse crashes only after they have occurred. Therefore they do not incur performance overhead. However, existing post-failure-process approaches still cannot reproduce many crashes in practice because of scalability issues and the object creation challenge. This paper proposes an automatic crash reproduction framework using collected crash stack traces. The proposed approach combines an efficient backward symbolic execution and a novel method sequence composition approach to generate unit test cases that can reproduce the original crashes without incurring additional runtime overhead. Our evaluation study shows that our approach successfully exploited 31 (59.6 percent) of 52 crashes in three open source projects. Among these exploitable crashes, 22 (42.3 percent) are useful reproductions of the original crashes that reveal the crash triggering bugs. A comparison study also demonstrates that our approach can effectively outperform existing crash reproduction approaches.

Index Terms—Crash reproduction, static analysis, symbolic execution, test case generation, optimization

1 Introduction

Software crash reproduction is the necessary first step for crash debugging. Unfortunately, manual crash reproduction is tedious and time consuming [15]. To assist crash reproduction, many techniques have been proposed [12], [20], [33], [38], [43], [52], [63] including record-replay approaches and post-failure-process approaches. Record-replay approaches [12], [43], [52] monitor and reproduce software executions by using software instrumentation techniques or special hardware that stores runtime information. Most record-replay approaches can reliably reproduce software crashes, but incur non-trivial performance overhead [12].

Different from record-replay approaches, post-failure-process approaches [20], [33], [38], [49], [63] analyse crashes only after they have occurred. Since post-failure-process approaches only use the crash data collected by the bug or crash reporting systems [4], [5], [6], [7], [14], [30], [51], they do not incur performance overhead to program executions. The purpose of post-failure-process approaches varies from explaining program failures [20], [38] to reproducing program failures [33], [63]. For failure explanation approaches, they try to explain the cause of a crash by analyzing the data-flow or crash condition information. For failure reproduction approaches, they try to reproduce a crash by synthesizing the original crash execution. However, the efficiency of existing failure reproduction approaches is not satisfactory due to scalability issues such as the *path explosion problem* [16], [31] where the number of potential paths to analyze grows exponentially to the number of conditional blocks involved. In addition, crashes for object-oriented programs cannot be effectively reproduced by these approaches because of the *object creation challenge* [60] where the desired object states cannot be achieved because non-public fields are not directly modifiable.

This paper proposes a Stack-Trace-based Automatic crash Reproduction framework (STAR), which automatically reproduces crashes using crash stack traces. STAR is a post-failure-process approach as it tries to reproduce a crash only after it has occurred. Compared to existing post-failure-process approaches, STAR has two major advantages: 1) STAR introduces several effective optimizations which can greatly boost the efficiency of the crash precondition computation process; and 2) Unlike existing failure reproduction approaches, STAR supports the reproduction of crashes for object-oriented programs using a novel method sequence composition approach. Therefore, the applicability of STAR is greatly broadened.

To reproduce a reported crash, STAR first performs a backward symbolic execution to identify the preconditions for triggering the target crash. Using a novel method sequence composition approach, STAR creates a test case that can generate test inputs satisfying the computed crash triggering preconditions. Since STAR uses the crash stack trace to reproduce a crash, its goal is to create unit test cases that can crash at the same location as the original crash and produce crash stack traces that are as close to the original crash stack trace as possible. The generated unit test cases can then be executed to reproduce the original crash and help reveal the underlying bug that caused this crash.

We have implemented STAR for the Java language and evaluated it with 52 real world crashes collected from three open source projects. STAR successfully exploited 31

- The authors are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {ning, hunkim}@cse.ust.hk.

Manuscript received 4 Dec. 2012; revised 16 Sept. 2014; accepted 5 Oct. 2014. Date of publication 13 Oct. 2014; date of current version 4 Feb. 2015. Recommended for acceptance by M. Dwyer. For information on obtaining reprints of this article, please send e-mail to reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2014.2363469

N. Chen and Kim, TSE 2015.

STAR (Stack Traced based Automatic crash Reproduction) uses backward Symbolic Execution triggering crash preconditions

Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Preconditions

```
v2.size == 0
v2 == not null
v1 == not null
v2 instanceOf(Ljava/util/HashMap)
v1 instanceOf(Lorg/apache/commons/collections/map/TransformedMap)
```

Test Case

```
public void test0() throws Throwable {
   java.util.HashMap v1 = new java.util.HashMap();
   java.util.HashMap v2 = new java.util.HashMap();
   org.apache.commons.collections.map.TransformedMap v3 =
         (org.apache.commons.collections.map.TransformedMap)
             org.apache.commons.collections.map.TransformedMap.decorate((java.util.Map) v2,
                 (org.apache.commons.collections.Transformer) null,
                         (org.apache.commons.collections.Transformer) null);
   v3.putAll((java.util.Map) v1);
}
```

Stack Trace based Crash Replication

Symbolic Execution

STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution

Ning Chen and Sunghun Kim, *Member, IEEE*

Abstract—Software crash reproduction is the necessary first step for debugging. Unfortunately, crash reproduction is often labor intensive. To automate crash reproduction, many techniques have been proposed including record-replay and post-failure-process approaches. Record-replay approaches can reliably replay recorded crashes, but they incur substantial performance overhead to program executions. Alternatively, post-failure-process approaches analyse crashes only after they have occurred. Therefore they do not incur performance overhead. However, existing post-failure-process approaches still cannot reproduce many crashes in practice because of scalability issues and the object creation challenge. This paper proposes an automatic crash reproduction framework using collected crash stack traces. The proposed approach combines an efficient backward symbolic execution and a novel method sequence composition approach to generate unit test cases that can reproduce the original crashes without incurring additional runtime overhead. Our evaluation study shows that our approach successfully exploited 31 (59.6 percent) of 52 crashes in three open source projects. Among these exploitable crashes, 22 (42.3 percent) are useful reproductions of the original crashes that reveal the crash triggering bugs. A comparison study also demonstrates that our approach can effectively outperform existing crash reproduction approaches.

Index Terms—Crash reproduction, static analysis, symbolic execution, test case generation, optimization

1 Introduction

Software crash reproduction is the necessary first step for crash debugging. Unfortunately, manual crash reproduction is tedious and time consuming [15]. To assist crash reproduction, many techniques have been proposed [12], [20], [33], [38], [43], [52], [63] including record-replay approaches and post-failure-process approaches. Record-replay approaches [12], [43], [52] monitor and reproduce software executions by using software instrumentation techniques or special hardware that stores runtime information. Most record-replay approaches can reliably reproduce software crashes, but incur non-trivial performance overhead [12].

Different from record-replay approaches, post-failure-process approaches [20], [33], [38], [49], [63] analyse crashes only after they have occurred. Since post-failure-process approaches only use the crash data collected by the bug or crash reporting systems [4], [5], [6], [7], [14], [30], [51], they do not incur performance overhead to program executions. The purpose of post-failure-process approaches varies from explaining program failures [20], [38] to reproducing program failures [33], [63]. For failure explanation approaches, they try to explain the cause of a crash by analyzing the data-flow or crash condition information. For failure reproduction approaches, they try to reproduce a crash by synthesizing the original crash execution. However, the efficiency of existing failure reproduction approaches is not satisfactory due to scalability issues such as the *path explosion problem* [16], [31] where the number of potential paths to analyze grows exponentially to the number of conditional blocks involved. In addition, crashes for object-oriented programs cannot be effectively reproduced by these approaches because of the *object creation challenge* [60] where the desired object states cannot be achieved because non-public fields are not directly modifiable.

This paper proposes a Stack-Trace-based Automatic crash Reproduction framework (STAR), which automatically reproduces crashes using crash stack traces. STAR is a post-failure-process approach as it tries to reproduce a crash only after it has occurred. Compared to existing post-failure-process approaches, STAR has two major advantages: 1) STAR introduces several effective optimizations which can greatly boost the efficiency of the crash precondition computation process; and 2) Unlike existing failure reproduction approaches, STAR supports the reproduction of crashes for object-oriented programs using a novel method sequence composition approach. Therefore, the applicability of STAR is greatly broadened.

To reproduce a reported crash, STAR first performs a backward symbolic execution to identify the preconditions for triggering the target crash. Using a novel method sequence composition approach, STAR creates a test case that can generate test inputs satisfying the computed crash triggering preconditions. Since STAR uses the crash stack trace to reproduce a crash, its goal is to create unit test cases that can crash at the same location as the original crash and produce crash stack traces that are as close to the original crash stack trace as possible. The generated unit test cases can then be executed to reproduce the original crash and help reveal the underlying bug that caused this crash.

We have implemented STAR for the Java language and evaluated it with 52 real world crashes collected from three open source projects. STAR successfully exploited 31

• *The authors are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {ning, hunkim}@cse.ust.hk.*

Manuscript received 4 Dec. 2012; revised 16 Sept. 2014; accepted 5 Oct. 2014. Date of publication 13 Oct. 2014; date of current version 4 Feb. 2015.
Recommended for acceptance by M. Dwyer.
For information on obtaining reprints of this article, please send e-mail to reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2363469

N. Chen and Kim, TSE 2015.

Advantages:

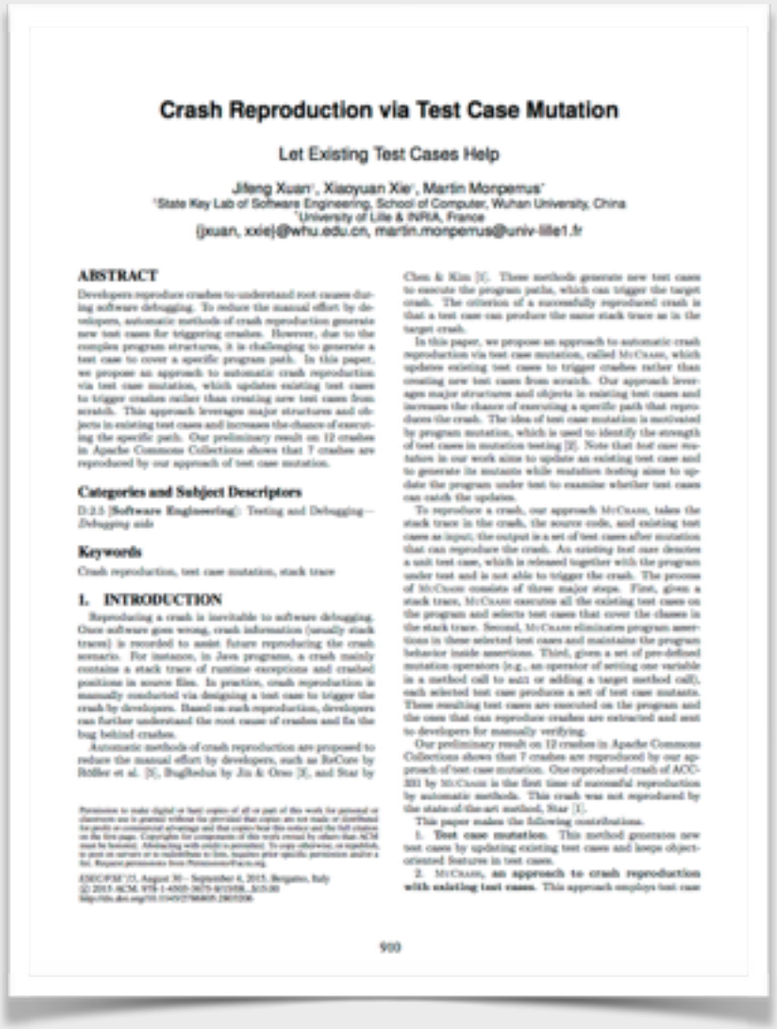
1) Better than Randoop (random testing)
2) Better than BugRedux

Disadvantages:

1) Crashes with environmental dependencies (e.g, external files) are not replicable
2) Path explosion
3) SMT solver limitations

Stack Trace based Crash Replication

Mutation Analysis



Crash Reproduction via Test Case Mutation

Let Existing Test Cases Help

Jifeng Xuan*, Xiaoyuan Xie*, Martin Monperrus†
*State Key Lab of Software Engineering, School of Computer, Wuhan University, China
†University of Lille & INRIA, France
{jxuan, xxie}@whu.edu.cn, martin.monperrus@univ-lille1.fr

ABSTRACT

Developers reproduce crashes to understand root causes during software debugging. To reduce the manual effort by developers, automatic methods of crash reproduction generate new test cases for triggering crashes. However, due to the complex program structures, it is challenging to generate a test case to cover a specific program path. In this paper, we propose an approach to automatic crash reproduction via test case mutation, which updates existing test cases to trigger crashes rather than creating new test cases from scratch. This approach leverages major structures and objects in existing test cases and increases the chance of executing the specific path. Our preliminary result on 12 crashes in Apache Commons Collections shows that 7 crashes are reproduced by our approach of test case mutation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

Keywords

Crash reproduction, test case mutation, stack trace

1. INTRODUCTION

Reproducing a crash is inevitable to software debugging. Once software goes wrong, crash information (usually stack traces) is recorded to assist future reproducing the crash scenario. For instance, in Java programs, a crash mainly contains a stack trace of runtime exceptions and crashed positions in source files. In practice, crash reproduction is manually conducted via designing a test case to trigger the crash by developers. Based on such reproduction, developers can further understand the root cause of crashes and fix the bug behind crashes.

Automatic methods of crash reproduction are proposed to reduce the manual effort by developers, such as ReCore by Rößler et al. [8], BugRedux by Jin & Orso [3], and Star by Chen & Kim [1]. These methods generate new test cases to execute the program paths, which can trigger the target crash. The criterion of a successfully reproduced crash is that a test case can produce the same stack trace as in the target crash.

In this paper, we propose an approach to automatic crash reproduction via test case mutation, called MuCrash, which updates existing test cases to trigger crashes rather than creating new test cases from scratch. Our approach leverages major structures and objects in existing test cases and increases the chance of executing a specific path that reproduces the crash. The idea of test case mutation is motivated by program mutation, which is used to identify the strength of test cases in mutation testing [2]. Note that test case mutation in our work aims to update an existing test case and to generate its mutants while mutation testing aims to update the program under test to examine whether test cases can catch the updates.

To reproduce a crash, our approach MuCrash, takes the stack trace in the crash, the source code, and existing test cases as input; the output is a set of test cases after mutation that can reproduce the crash. An existing test case denotes a unit test case, which is released together with the program under test and is not able to trigger the crash. The process of MuCrash consists of three major steps. First, given a stack trace, MuCrash executes all the existing test cases on the program and selects test cases that cover the classes in the stack trace. Second, MuCrash eliminates program assertions in these selected test cases and maintains the program behavior inside assertions. Third, given a set of pre-defined mutation operators (e.g., an operator of setting one variable in a method call to null or adding a target method call), each selected test case produces a set of test case mutants. These resulting test cases are executed on the program and the ones that can reproduce crashes are extracted and sent to developers for manually verifying.

Our preliminary result on 12 crashes in Apache Commons Collections shows that 7 crashes are reproduced by our approach of test case mutation. One reproduced crash of ACC-331 by MuCrash is the first time of successful reproduction by automatic methods. This crash was not reproduced by the state-of-the-art method, Star [1].

This paper makes the following contributions.

1. **Test case mutation.** This method generates new test cases by updating existing test cases and keeps object-oriented features in test cases.
2. MuCrash, **an approach to crash reproduction with existing test cases.** This approach employs test case

ASE'15, August 30 – September 6, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803208

J. Xuan et al., ESEC/FSE 2015.

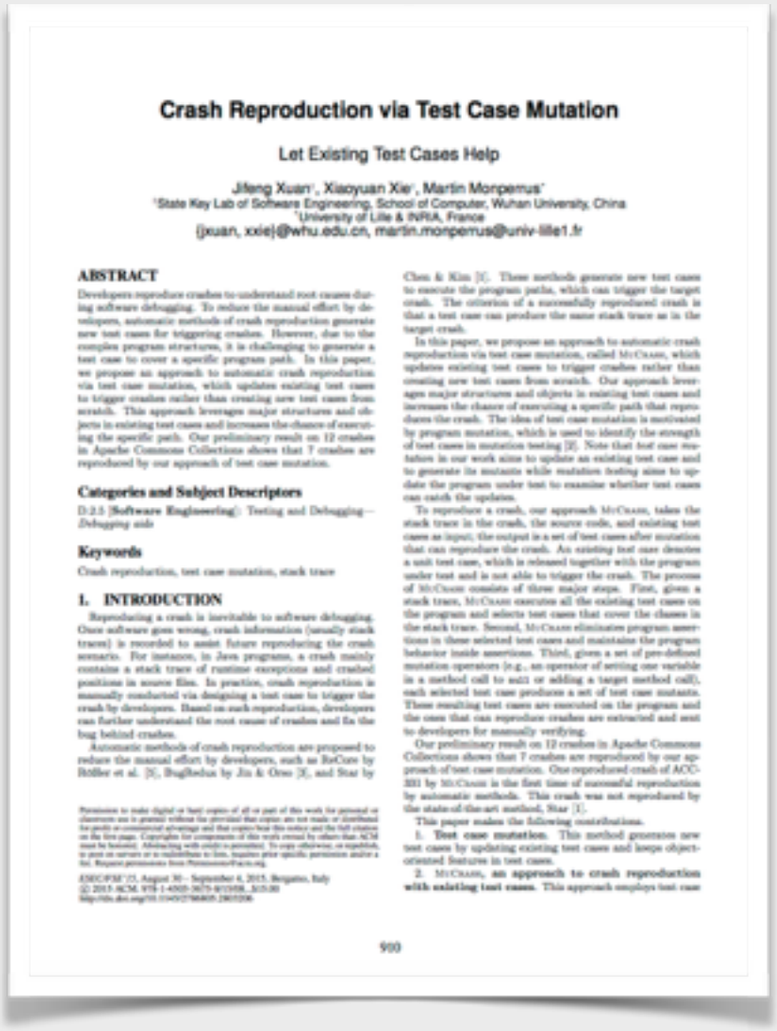
```
@Test
public void test(){
    Boolean boolean0 = false;
    String string0 = BooleanUtils.toStringOnOff(boolean0);
    assertNotNull(string0);
    assertEquals("off", string0);
}
```

MuCrash

```
@Test
public void test(){
    Boolean boolean0 = false;
    String string0 = BooleanUtils.toStringOnOff(null);
    assertNotNull(string0);
    assertEquals("off", string0);
}
```

Stack Trace based Crash Replication

Mutation Analysis



Crash Reproduction via Test Case Mutation

Let Existing Test Cases Help

Jifeng Xuan*, Xiaoyuan Xie*, Martin Monperrus†
*State Key Lab of Software Engineering, School of Computer, Wuhan University, China
†University of Lille & INRIA, France
{jxuan, xxie}@whu.edu.cn, martin.monperrus@univ-lille1.fr

ABSTRACT

Developers reproduce crashes to understand root causes during software debugging. To reduce the manual effort by developers, automatic methods of crash reproduction generate new test cases for triggering crashes. However, due to the complex program structures, it is challenging to generate a test case to cover a specific program path. In this paper, we propose an approach to automatic crash reproduction via test case mutation, which updates existing test cases to trigger crashes rather than creating new test cases from scratch. This approach leverages major structures and objects in existing test cases and increases the chance of executing the specific path that reproduces the crash. The idea of test case mutation is motivated by program mutation, which is used to identify the strength of test cases in mutation testing [2]. Note that test case mutation is different from program mutation. Our preliminary result on 12 crashes in Apache Commons Collections shows that 7 crashes are reproduced by our approach of test case mutation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

Keywords

Crash reproduction, test case mutation, stack trace

1. INTRODUCTION

Reproducing a crash is inevitable to software debugging. Once software goes wrong, crash information (usually stack traces) is recorded to assist future reproducing the crash scenario. For instance, in Java programs, a crash mainly contains a stack trace of runtime exceptions and crashed positions in source files. In practice, crash reproduction is manually conducted via designing a test case to trigger the crash by developers. Based on such reproduction, developers can further understand the root cause of crashes and fix the bug behind crashes.

Automatic methods of crash reproduction are proposed to reduce the manual effort by developers, such as ReCore by Rößler et al. [5], BugRedux by Jin & Orso [3], and Star by Chen & Kim [1]. These methods generate new test cases to execute the program paths, which can trigger the target crash. The criterion of a successfully reproduced crash is that a test case can produce the same stack trace as in the target crash.

In this paper, we propose an approach to automatic crash reproduction via test case mutation, called MuCrash, which updates existing test cases to trigger crashes rather than creating new test cases from scratch. Our approach leverages major structures and objects in existing test cases and increases the chance of executing a specific path that reproduces the crash. The idea of test case mutation is motivated by program mutation, which is used to identify the strength of test cases in mutation testing [2]. Note that test case mutation is different from program mutation. Program mutation aims to update the program under test to generate its mutants while mutation testing aims to update the program under test to examine whether test cases can catch the updates.

To reproduce a crash, our approach MuCrash, takes the stack trace in the crash, the source code, and existing test cases as input, the output is a set of test cases after mutation that can reproduce the crash. An existing test case denotes a unit test case, which is released together with the program under test and is not able to trigger the crash. The process of MuCrash consists of three major steps. First, given a stack trace, MuCrash executes all the existing test cases on the program and selects test cases that cover the classes in the stack trace. Second, MuCrash eliminates program assertions in these selected test cases and maintains the program behavior inside assertions. Third, given a set of pre-defined mutation operators (e.g., an operator of setting one variable in a method call to null or adding a target method call), each selected test case produces a set of test case mutants. These resulting test cases are executed on the program and the ones that can reproduce crashes are extracted and sent to developers for manually verifying.

Our preliminary result on 12 crashes in Apache Commons Collections shows that 7 crashes are reproduced by our approach of test case mutation. One reproduced crash of ACC-331 by MuCrash is the first time of successful reproduction by automatic methods. This crash was not reproduced by the state-of-the-art method, Star [1].

This paper makes the following contributions.

1. **Test case mutation**. This method generates new test cases by updating existing test cases and keeps object-oriented features in test cases.
2. MuCrash, **an approach to crash reproduction with existing test cases**. This approach employs test case

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803206

J. Xuan et al., ESEC/FSE 2015.

Advantages:

1) Replicate some crashes not replicable by STAR
2) No solver is used

Disadvantages:

1) Leads to a large number of unnecessary test cases
2) Crashes requiring method sequences (not included in the original test case) are not reproduced

What about SBST Unit Test Tools?

Are they competitive if relying on Stack Traces only?

Why Unit Test Tools?

Target Crash

Bug Name: ACC-48

Library: Apache Commons Collection

Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Exception Name

Root Cause of the Exception

https://issues.apache.org/jira/browse/COLLECTIONS-48

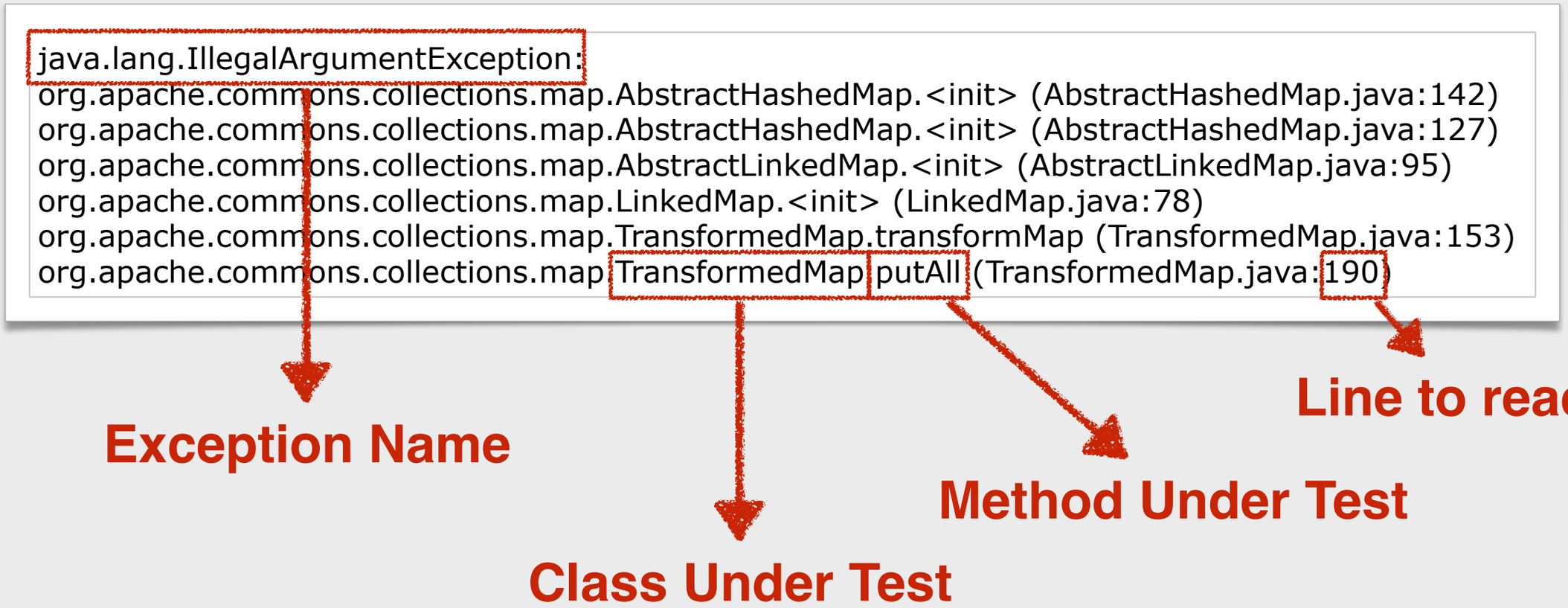
Why Unit Test Tools?

Target Crash

Bug Name: ACC-48

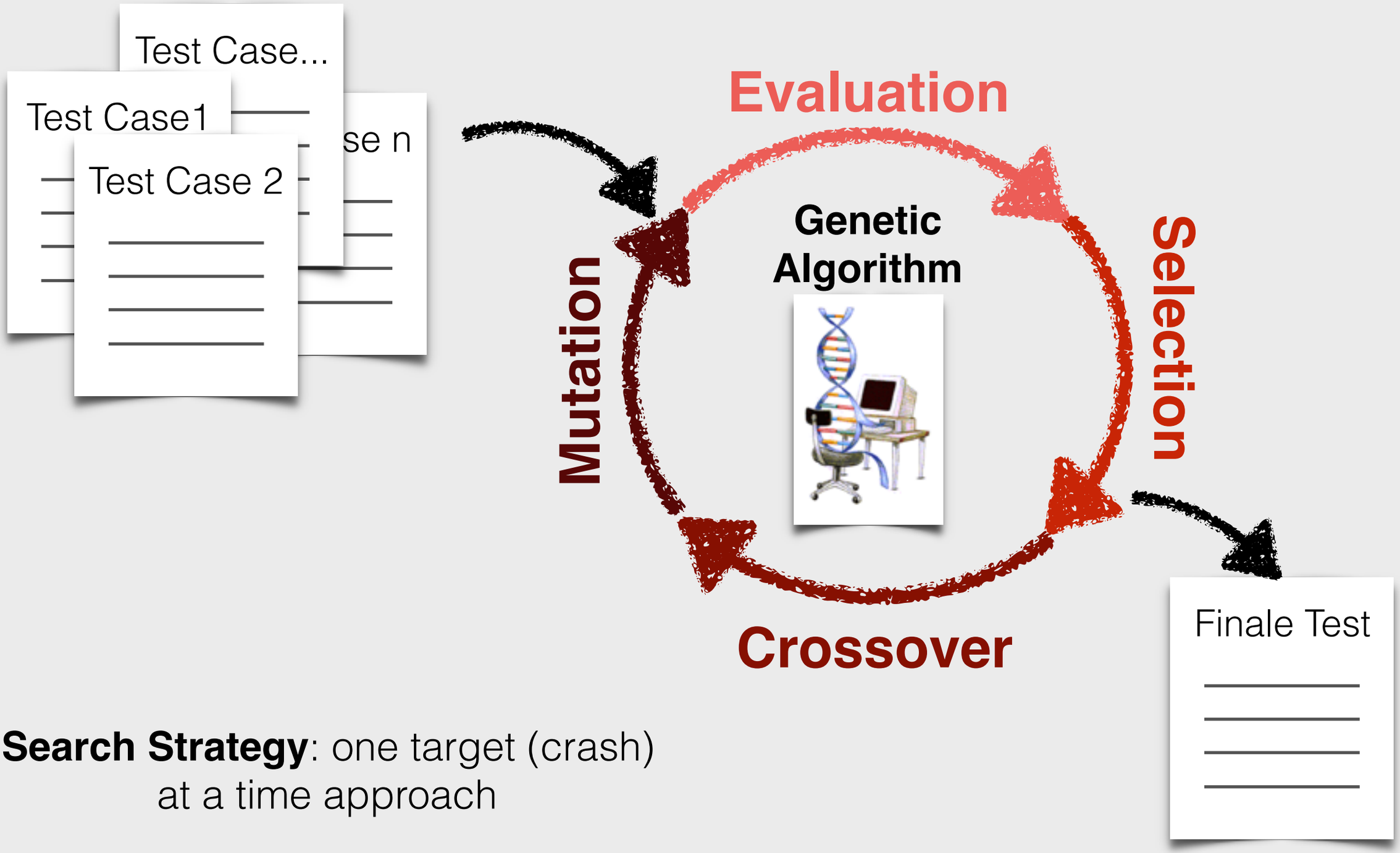
Library: Apache Commons Collection

Stack Trace



https://issues.apache.org/jira/browse/COLLECTIONS-48

EVOSUITE

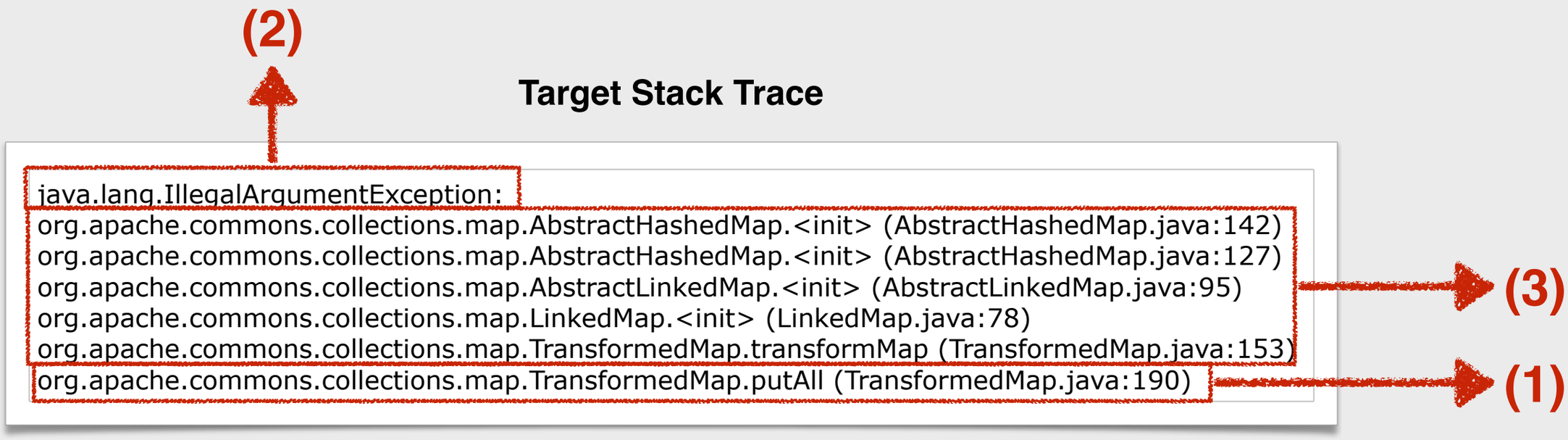


Search Strategy: one target (crash) at a time approach

Fitness Function

Main Conditions to Satisfy

1) the line (statement) where the exception is thrown has to be covered
2) the target exception has to be thrown
3) the generated stack trace must be as similar to the original one as possible.



$f(t)$ = 3 x line_coverage + 2 x exception_coverage + trace_similarity

(1) (2) (3)

Fitness Function

$$f(t) = 3 \times \text{line\_coverage} + 2 \times \text{exception\_coverage} + \text{trace\_similarity}$$

(1) **(2)** **(3)**

1) **line_coverage** = approach_level + branch_distance

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Fitness Function

$$f(t) = 3 \times \text{line\_coverage} + 2 \times \text{exception\_coverage} + \text{trace\_similarity}$$

(1) **(2)** **(3)**

1) **line_coverage** = approach_level + branch_distance

2) **exception_coverage** = 0 if the target exception in thrown; 1 otherwise

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Fitness Function

$$f(t) = 3 \times \text{line\_coverage} + 2 \times \text{exception\_coverage} + \text{trace\_similarity}$$

(1) **(2)** **(3)**

1) **line_coverage** = approach_level + branch_distance

2) **exception_coverage** = 0 if the target exception in thrown; 1 otherwise

3) **trace_similarity** = cumulative differences with target trace elements

Trace Elements

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Trace Similarity

Two trace elements are equal iff:
1) same class name
2) same method name
3) same line

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Generated Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:141)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:48)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:31)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:72)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:148)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Trace Similarity

Two trace elements are equal iff:
1) same class name **(distance = 0)**
2) same method name
3) same line

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Generated Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:141)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:48)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:31)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:72)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:148)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Trace Similarity

Two trace elements are equal iff:

1) same class name **(distance = 0)**
2) same method name **(distance = 0)**
3) same line

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Generated Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:141)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:48)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:31)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:72)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:148)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Trace Similarity

Two trace elements are equal iff:

1) same class name **(distance = 0)**
2) same method name **(distance = 0)**
3) same line **(distance = |153 - 148| / (1+ |153 - 148|) = 0.83)**

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Generated Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:141)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:48)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:31)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:72)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:148)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Trace Similarity

Two trace elements are equal iff:

1) same class name **(distance = 0)**
2) same method name **(distance = 0)**
3) same line **(distance = |153 - 148| / (1+ |153 - 148|) = 0.83)**

Target Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:142)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:127)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:95)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:78)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:153)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

Generated Stack Trace

```
java.lang.IllegalArgumentException:
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:141)
org.apache.commons.collections.map.AbstractHashedMap.<init> (AbstractHashedMap.java:48)
org.apache.commons.collections.map.AbstractLinkedMap.<init> (AbstractLinkedMap.java:31)
org.apache.commons.collections.map.LinkedMap.<init> (LinkedMap.java:72)
org.apache.commons.collections.map.TransformedMap.transformMap (TransformedMap.java:148)
org.apache.commons.collections.map.TransformedMap.putAll (TransformedMap.java:190)
```

0.5
0.99
0.98
0.85
0.83
Tot = 4.15

Pilot Study

Empirical Evaluation

Context: 10 real bugs from Apache Commons Collections

Bug ID	Version	Exception	Priority
ACC-4	2.0	NullPointer	Major
ACC-28	2.0	NullPointer	Major
ACC-35	2.0	UnsupportedOperation	Major
ACC-48	3.1	IllegalArgument	Major
ACC-53	3.1	ArrayIndexOutOfBound	Major
ACC-70	3.1	NullPointer	Major
ACC-77	3.1	IllegalState	Major
ACC-104	3.1	ArrayIndexOutOfBound	Major
ACC-331	3.2	NullPointer	Minor
ACC-377	3.2	NullPointer	Minor

Used in:

N. Chen and Kim, TSE 2015.

J. Xuan et al., ESEC/FSE 2015.

Experimented algorithms:

- EvoSuite + our fitness function (30 independent runs)
- STAR (Symbolic execution)
- MuCrash (Mutation analysis)

Results

Bug ID	% Successful Replication	STAR	MuCrash
ACC-4	30/30	YES	YES
ACC-28	30/30	YES	YES
ACC-35	30/30	YES	YES
ACC-48	30/30	YES	YES
ACC-53	28/30	YES	NO
ACC-70	30/30	NO	NO
ACC-77	30/30	YES	NO
ACC-104	0/30	YES	YES
ACC-331	10/30	NO	YES
ACC-377	0/30	NO	NO

Results

Bug ID	% Successful Replication	STAR	MuCrash
ACC-4	30/30	YES	YES
ACC-28	30/30	YES	YES
ACC-35	30/30	YES	YES
ACC-48	30/30	YES	YES
ACC-53	**28/30**	**YES**	**NO**
ACC-70	**30/30**	**NO**	**NO**
ACC-77	**30/30**	**YES**	**NO**
ACC-104	**0/30**	**YES**	**YES**
ACC-331	**10/30**	**NO**	**YES**
ACC-377	0/30	NO	NO

Our solution replicated 8/10 bugs

STAR replicated 7/10 bugs

MuCrash replicated 6/10 bugs

Results

Bug ID	% Successful Replication	STAR	MuCrash
ACC-4	30/30	YES	YES
ACC-28	30/30	YES	YES
ACC-35	30/30	YES	YES
ACC-48	30/30	YES	YES
ACC-53	**28/30**	**YES**	**NO**
ACC-70	**30/30**	**NO**	**NO**
ACC-77	**30/30**	**YES**	**NO**
ACC-104	**0/30**	**YES**	**YES**
ACC-331	**10/30**	**NO**	**YES**
ACC-377	0/30	NO	NO

Replicable by our SBST solution only

ACC-70

Target Stack Trace

```
Exception in thread "main" java.lang.NullPointerException at
org.apache.commons.collections.list.TreeList$TreeListIterator.previous (TreeList.java:841)
at java.util.Collections.get(Unknown Source)
at java.util.Collections.iteratorBinarySearch(Unknown Source)
at java.util.Collections.binarySearch(Unknown Source)
at utils.queue.QueueSorted.put(QueueSorted.java:51)
at framework.search.GraphSearch.solve(GraphSearch.java:53)
at search.informed.BestFirstSearch.solve(BestFirstSearch.java:20)
at Hlavni.main(Hlavni.java:66)
```

ACC-70

Target Stack Trace

```
Exception in thread "main" java.lang.NullPointerException at
org.apache.commons.collections.list.TreeList$TreeListIterator.previous (TreeList.java:841)
at java.util.Collections.get(Unknown Source)
at java.util.Collections.iteratorBinarySearch(Unknown Source)
at java.util.Collections.binarySearch(Unknown Source)
at utils.queue.QueueSorted.put(QueueSorted.java:51)
at framework.search.GraphSearch.solve(GraphSearch.java:53)
at search.informed.BestFirstSearch.solve(BestFirstSearch.java:20)
at Hlavni.main(Hlavni.java:66)
```

Test generated by our solution

```
public void test0()  throws Throwable  {
    TreeList treeList0 = new TreeList();
    treeList0.add((Object) null);
    TreeList.TreeListIterator treeList_TreeListIterator0 = new
                TreeList.TreeListIterator(treeList0, 732);
    // Undeclared exception!
    treeList_TreeListIterator0.previous();
}
```

ACC-70

Target Stack Trace

```
Exception in thread "main" java.lang.NullPointerException at
org.apache.commons.collections.list.TreeList$TreeListIterator.previous (TreeList.java:841)
at java.util.Collections.get(Unknown Source)
at java.util.Collections.iteratorBinarySearch(Unknown Source)
at java.util.Collections.binarySearch(Unknown Source)
at utils.queue.QueueSorted.put(QueueSorted.java:51)
at framework.search.GraphSearch.solve(GraphSearch.java:53)
at search.informed.BestFirstSearch.solve(BestFirstSearch.java:20)
at Hlavni.main(Hlavni.java:66)
```

Test generated by our solution

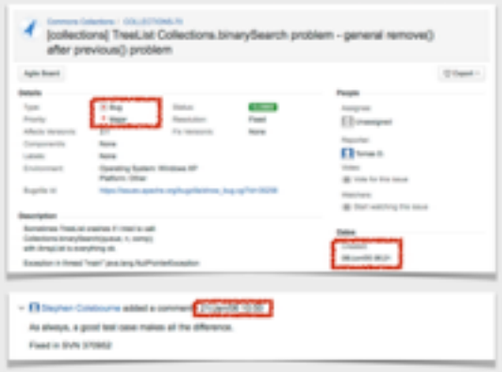
```
public void test0()  throws Throwable  {
    TreeList treeList0 = new TreeList();
    treeList0.add((Object) null);
    TreeList.TreeListIterator treeList_TreeListIterator0 = new
                TreeList.TreeListIterator(treeList0, 732);
    // Undeclared exception!
    treeList_TreeListIterator0.previous();
}
```

Affected Code

```
public Object previous() {
   ...
   if (next == null) {
        next = parent.root.get(nextIndex - 1);
    } else {
        next = next.previous();
   }
   Object value = next.getValue();
   ...
 }
}
```

if “parent” is null, this code generates an exception

Cost of Bug Fixings



Major Bug for Apache Collections

Created on June 2005

Solved on January 2006

https://issues.apache.org/jira/browse/COLLECTIONS-70

Stack Trace based Crash Replication

Symbolic Execution

N. Chen and Kim, TSE 2015.

STAR (Stack Traced based Automatic crash Reproduction) uses backward Symbolic Execution triggering crash preconditions

Stack Trace

Preconditions

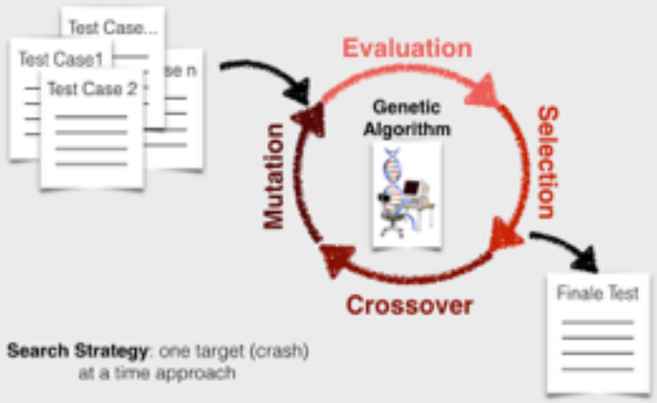
Test Case

Stack Trace based Crash Replication

Mutation Analysis

J. Xuan et al., ESEC/FSE 2015.

EVOSUITE



Search Strategy: one target (crash) at a time approach

Fitness Function

$f(t)$ = 3 x line_coverage + 2 x exception_coverage + trace_similarity
(1) (2) (3)

1) **line_coverage** = approach_level + branch_distance

2) **exception_coverage** = 0 if the target exception in thrown; 1 otherwise

3) **trace_similarity** = cumulative differences with target trace elements

Target Stack Trace

Trace Elements

Empirical Evaluation

Context: 10 real bugs from Apache Commons Collections

Bug ID	Version	Exception	Priority
ACC-4	2.0	NullPointer	Major
ACC-28	2.0	NullPointer	Major
ACC-35	2.0	UnsupportedOperation	Major
ACC-48	3.1	IllegalArgument	Major
ACC-53	3.1	ArrayIndexOutOfBound	Major
ACC-70	3.1	NullPointer	Major
ACC-77	3.1	IllegalState	Major
ACC-104	3.1	ArrayIndexOutOfBound	Major
ACC-331	3.2	NullPointer	Minor
ACC-377	3.2	NullPointer	Minor

Used in:

N. Chen and Kim, TSE 2015.

J. Xuan et al., ESEC/FSE 2015.

Experimented algorithms:
- EvoSuite + our fitness function (30 independent runs)
- STAR (Symbolic execution)
- MuCrash (Mutation analysis)

Results

Bug ID	% Successful Replication	STAR	MuCrash
ACC-4	30/30	YES	YES
ACC-28	30/30	YES	YES
ACC-35	30/30	YES	YES
ACC-48	30/30	YES	YES
ACC-53	28/30	YES	NO
ACC-70	30/30	NO	NO
ACC-77	30/30	YES	NO
ACC-104	0/30	YES	YES
ACC-331	10/30	NO	YES
ACC-377	0/30	NO	NO

Our solution replicated 8/10 bugs

STAR replicated 7/10 bugs

MuCrash replicated 6/10 bugs

ACC-70

Target Stack Trace

Test generated by our solution

Affected Code

if "parent" is null, this code generates an exception

Thank you!
Q&A